Um acelerador em hardware para a cifra ChaCha20

Khalil Glevinski Queiroz de Santana
Universidade do Vale do Itajaí, Brasil
khalil.santana@edu.univali.br

Douglas Rossi de Melo
Universidade do Vale do Itajaí, Brasil
drm@univali.br

ABSTRACT

Current computer systems use encryption extensively, for example, in instant messaging, virtual private network protocols, and websites in general. However, employing encryption in a system can be computationally costly, resulting in lower throughput or higher latencies. One strategy to mitigate the costs imposed by the application of cryptography is the use of accelerators dedicated to the execution of this task. In this context, this paper presents the development of an encryption accelerator for the ChaCha20 stream cipher. The accelerator was prototyped on two FPGA SoC platforms, achieving a normalized speedup of 3.49 times compared to the software implementation.

KEYWORDS

Sistemas Embarcados, Aceleradores, FPGA, Criptografia, RFC 8439

1 INTRODUÇÃO

A palavra criptografia tem sua origem etimológica no grego antigo, significando "oculto" e "escrita". É a arte de escrever ou codificar mensagens secretas. Autores como Paar e Pelzl [1] relatam os primeiros usos de criptografia datando por volta de 2000AC no Egito Antigo. Desde então, a criptografia foi usada por diversas civilizações como, por exemplo, a Cifra de Cesar na Roma Antiga.

Apesar de sua origem antiga, a criptografia é relevante até os dias atuais, sendo aplicada em diversos contextos como para proteger os dados no *internet banking*, videochamadas, mensageiros instantâneos e afins. Em [2] é apresentado um estudo o qual concluiu que 85% de todo o tráfego na internet é criptografado.

Diversas áreas, não limitadas a criptografia, ao se depararem com uma tarefa computacionalmente custosa ou de baixo desempenho, optam por executar essa tarefa através de um componente de hardware dedicado, também conhecido como acelerador ou coprocessador.

Dentro deste contexto, este trabalho procura contribuir na área de aceleradores criptográficos, visando a criação de um acelerador voltado a esta finalidade, assim como a coleta de métricas de custo e desempenho referentes a seu uso em comparação a uma implementação em software.

Este artigo está divido em seis seções. A Seção 2 apresenta a fundamentação teórica deste trabalho, partindo de princípios de sistemas embarcados, criptografia e a cifra escolhida a ser acelerada. A Seção 3 apresenta os trabalhos relacionados. Na Seção 4 é descrito o desenvolvimento do acelerador. A Seção 5 apresenta as métricas de custo e desempenho do acelerador desenvolvido. Finalmente, a Seção 6 contêm as considerações finais deste artigo.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta conceitos sobre sistemas embarcados, criptografia e a cifra selecionada para a aceleração.

2.1 Sistemas Embarcados

Segundo Vahid e Givargis [3], sistemas computacionais estão em todos os lugares, desde servidores a computadores pessoais. Entretanto, uma parte significativa desses sistemas são utilizados para um propósito distinto e, por vezes, invisíveis ao usuário. Esses sistemas são conhecidos como sistemas embarcados, e possuem como características principais:

- Função única: geralmente executam apenas uma função, de maneira repetida;
- Fortemente limitado: podem ser particularmente mais restritivos em suas métricas de projeto, como tamanho, custo, desempenho e consumo energético;
- Reativo e em tempo real: precisam continuamente reagir a mudanças no ambiente, repetidamente, e produzir uma resposta dentro de um limite de tempo.

Sistemas embarcados podem então possuir uma série de métricas de projeto a serem alcançados, como por exemplo, um baixo custo unitário, alta vazão, baixo consumo energético, dentre outros [3].

O processador é o elemento principal de um sistema embarcado. Esse pode ser projetado por meio de linguagens de descrição de hardware (HDLs – Hardware Description Languages). Existem duas classes de processadores embarcados, os de propósito geral e os de propósito específico. Os processadores de propósito geral são sistemas digitais reprogramáveis, também conhecidos como CPUs (Central Processing Units), os quais são capazes de realizar uma grande gama de problemas computacionais. Já os processadores de propósito específico, também conhecidos como aceleradores ou coprocessadores, são especializados na resolução de problemas distintos, como por exemplo, criptografia [3].

2.2 Criptografia

A criptografia é a ciência de escrever segredos com o objetivo de esconder o significado de uma mensagem. Apesar de a criptografia estar comumente associada a computadores, sua existência precede os mesmos, sendo particularmente utilizada em contextos militares, diplomáticos e outros fins governamentais [4].

Dentre os principais objetivos da criptografia está a confidencialidade da mensagem e sua integridade, isto é, o conteúdo da mensagem a ser transmitida deve ser ininteligível para entidades sem a posse da chave criptográfica, assim como destinatários deverão ser capazes de determinar se a mensagem foi manipulada durante o trânsito [5].

Existem diversas subdivisões dentro da criptografia, em específico, a classificação de cada tipo de cifra em diversas categorias, como simétricas, assimétricas, baseadas em blocos ou em fluxo, assim como distintos modos de operação.

Para Alvarenga [5], a criptografia em fluxo é definida como um processo no qual uma chave inicial é utilizada para a criação de uma sequência de chaves (*key stream*), a qual por sua vez é combinada

com o texto claro utilizando a operação XOR, ou seja, os bits da *key stream* são combinados com os bits do texto claro, bit-a-bit, resultando no texto cifrado. O processo de decifragem é similar, utilizando a mesma chave inicial para derivar a *key stream* e, em seguida, combinar esta com o texto cifrado, bit-a-bit, utilizando a operação XOR.

Segundo Menezes et al. [4], as cifras de fluxo são geralmente mais rápidas do que cifras em blocos quando implementada em hardware, e possuem um projeto menos complexo em relação aos circuitos necessários. Além disso, este tipo de cifra pode ser necessário para determinadas finalidades, como por exemplo, em aplicações de telecomunicações.

2.3 ChaCha20

Em 2008 Daniel J. Bernstein propôs uma nova família de cifras de fluxo chamada ChaCha, baseada no seu trabalho na família Salsa de cifras [6]. Além disso, a construção desta cifra foi padronizada na RFC 8439 [7], como o tamanho do campo `nonce` e contador, que foram modificados para 32 e 96 bits, respectivamente, em contraste ao artigo original de Bernstein que utilizou de 64 bits para ambos os campos.

O algoritmo pode ser representado como operando em uma matriz $4x4$, onde cada elemento é uma palavra de 32 bits. Essa matriz pode ser observada na Figura 1, sendo inicializada da seguinte forma:

(1) Bits [0..127]: constante `expand 32-byte k` em ASCII.
(2) Bits [128..383]: chave de 256 bits.
(3) Bits [384..416]: contador e blocos de 32 bits.
(4) Bits [417..511]: `nonce` (number used once) de 96 bits.

Figura 1: Estado inicial da cifra ChaCha20 (RFC 8439)

O ChaCha utiliza de 4 adições, XORs e rotações para atualizar 4 palavras de 32 bits, utilizando a função Quarter Round (`Qround`) descrita no Quadro 1.

Quadro 1: Função Quarter Round (`Qround`)

```
a += b; d ^= a; d <<<= 16;
c += d; b ^= c; b <<<= 12;
a += b; d ^= a; d <<<= 8;
c += d; b ^= c; b <<<= 7;
```

Entretanto, o ChaCha possui 16 números inteiros, portanto a função (`Qround`) é invocada múltiplas vezes, utilizando elementos e colunas diferentes. Ao todo, o ChaCha20 possui 20 rodadas, alternando entre rodadas de coluna e rodadas diagonais, como ilustrado na Figura 2.

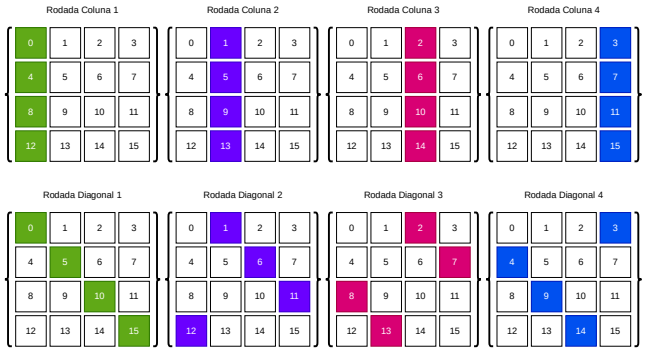

Figura 2: Rodadas de colunas e diagonais da cifra ChaCha

Após a aplicação da função `Qround`, em rodadas de colunas e diagonais, o estado resultante da matriz é somado ao estado anterior, compondo uma sequência de números a serem utilizadas uma única vez, conhecida como OTP (One-Time Pad). O OTP é então combinado bit-a-bit com o texto claro por meio da função XOR, resultando no texto cifrado como observado na Figura 3.

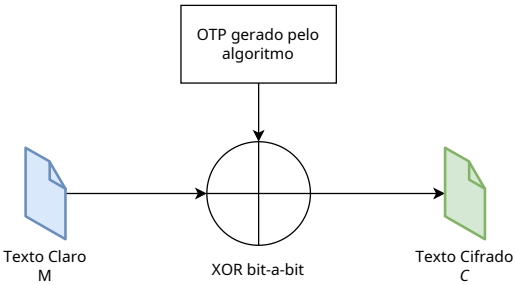

Figura 3: Etapa final da cifragem ChaCha20

No Quadro 2 é apresentado o pseudocódigo que descreve o algoritmo da cifra ChaCha20 [7]. Nota-se que a execução de 8 funções `Qround` equivale a duas rodadas completas. Sendo assim, o laço de atualização do estado interno é executado dez vezes para que se obtenha 20 rodadas.

Quadro 2: Pseudocódigo da cifra ChaCha20

```
inner_block (state):
 Qround(state, 0, 4, 8, 12)
 Qround(state, 1, 5, 9, 13)
 Qround(state, 2, 6, 10, 14)
 Qround(state, 3, 7, 11, 15)
 Qround(state, 0, 5, 10, 15)
 Qround(state, 1, 6, 11, 12)
 Qround(state, 2, 7, 8, 13)
 Qround(state, 3, 4, 9, 14)
 end
chacha20_block(key, counter, nonce):
 state = constants | key | counter | nonce
 initial_state = state
 for i=1 upto 10
 inner_block(state)
 end
 state += initial_state
 return serialize(state)
end
```

Fonte: RFC 8439 [7]

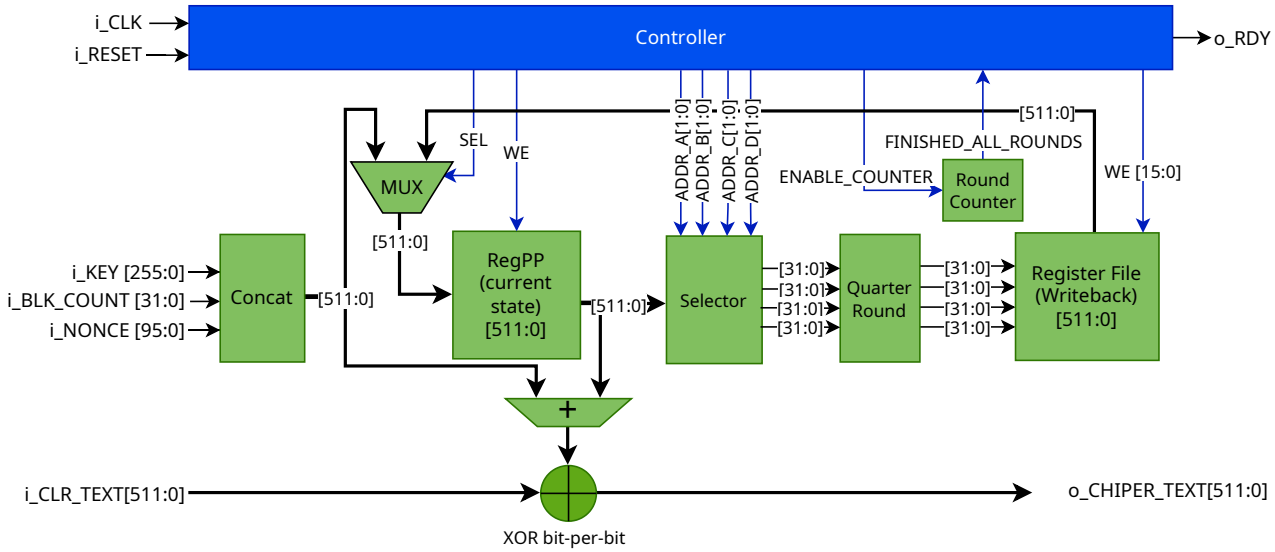

Figura 4: Diagrama de blocos do acelerador ChaCha20

3 TRABALHOS RELACIONADOS

Existem na literatura inúmeras implementações de aceleradores, incluindo diversos aceleradores criptográficos, os quais também optam por diferentes abordagens, métricas, ferramentas e linguagens. Na Tabela 1 é exibido um comparativo de cada trabalho relacionado citado nesta seção.

Sopran et al. [8] analisou métricas como vazão, consumo energético e área na implementação da cifra SIMON, incluindo abordagens de particionamento de hardware/software em uma FPGA. Isto é, a cifragem e decifragem possuem elementos computados tanto por lógica programável, quanto em software. Os autores observaram uma aceleração de 46,9% ao comparar sua implementação particionada com a versão em software.

Kumar et al. [9] utilizou um dispositivo ASIC para acelerar a cifra NORX, otimizando o algoritmo original para uma implementação em hardware mais eficiente. Os autores identificaram blocos que poderiam ser reutilizados na execução da cifra. Tais mudanças resultaram em um ganho de 44,8% e ocupando 18,01% menos área ao comparar a versão otimizada NORX com a versão original.

O trabalho de Shaher et al. [10] apresenta um acelerador para a cifra ASCON, com foco na proteção de imagens capturadas por câmeras de vigilância IoT. Ao realizar testes comparativos, os autores obtiveram uma aceleração de 13,34x se comparado a uma implementação em software no processador ARM Cortex-A9.

Por fim, Semenov [11] descreve as características que tornam a cifra ChaCha20 interessante para a aceleração em hardware. O autor otimizou o uso de recursos lógicos ao implementar funções menos utilizadas (adição e XOR) em software. A solução em hardware apresentou uma aceleração de 3,61x em comparação ao software.

Dentre os trabalhos relacionados, Semenov [11] se destacou como a mais similar a este trabalho. No entanto, existem diferenças significativas entre ambas, desde seu escopo à arquitetura abordada.

Este trabalho opta por realizar todas as operações pertinentes à cifragem e decifragem em hardware, enquanto [11] realiza determinadas operações em software, como a adição dos estados internos da cifra e a operação XOR entre o texto claro e o OTP. Além disso, o autor contempla outros componentes não desenvolvidos neste trabalho, como a comunicação entre o processador e a lógica programável, inviabilizando uma comparação direta das métricas de desempenho entre as soluções.

Tabela 1: Comparativo dos trabalhos relacionados

Trabalho	Algoritmo	Plataforma	Linguagem
[8]	SIMON	Altera Cyclone II	VHDL
[9]	NORX	ASIC TSMC 65nm	VHDL
[10]	ASCON	Xilinx Zynq 7000	VHDL
[11]	ChaCha20	Altera Cyclone V	Verilog
Este trabalho	ChaCha20	Altera Cyclone V	VHDL
Este trabalho	ChaCha20	Xilinx Zynq 7000	VHDL

4 DESENVOLVIMENTO

A Figura 4 apresenta o diagrama de blocos do acelerador desenvolvido. Nesta seção são descritas as características de cada componente, como sua finalidade, entradas, saídas e funcionalidades específicas.

Concatenador

O bloco `Concat`, apresentado na Figura 5, tem como propósito concatenar as diversas entradas da cifra, como o contador do bloco atual (`i_BLK_COUNT`), a chave (`i_KEY`) e o nonce (`i_NONCE`). Estas entradas representam 3/4 da matriz de estado inicial, com os 128 bits restantes correspondendo à constante "`expand 32-byte k`".

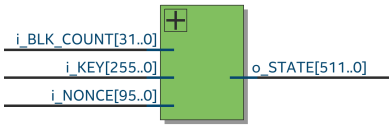

Figura 5: Concatenador (`Concat`)

Multiplexador de Entradas

O `MUX` tem como sua única função chavear entre o estado inicial (`i_DATA_FROM_CONCAT`) e o estado da rodada anterior (`i_DATA_FROM_LAST_ROUND`), para a entrada do registrador paralelo-paralelo.

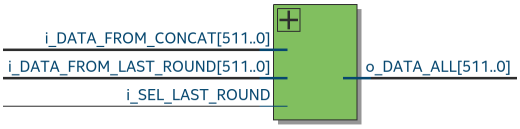

Figura 6: Multiplexador de Entradas (`MUX`)

Registrador Paralelo-Paralelo

O bloco `RegPP` (Figura 7) se trata de um registrador de 512 bits responsável por armazenar o estado atual da cifra, normalmente representada como uma matriz $4x4$ com elementos de 32 bits, bem como ser atualizado conforme as rodadas são realizadas.

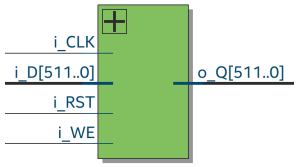

Figura 7: Registrador Paralelo-Paralelo (`RegPP`)

Seletor de Elementos

O componente `Selector` (Figura 8) corresponde a um conjunto de multiplexadores cujas entradas são os 512 bits providos pelo `RegPP` (entrada `i_D`) e quatro entradas de endereço (`i_ADDR_A`, `i_ADDR_B`, `i_ADDR_C`, `i_ADDR_D`) para controlar o seletor desses multiplexadores. Sua saída corresponde a quatro sinais de 32 bits, representando os elementos selecionados (requeridos) para aquela rodada na operação Quarter Round.

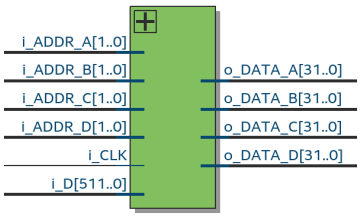

Figura 8: Seletor de Elementos (`Selector`)

Quarter Round

A função `QuarterRound` (Figura 8) possui quatro entradas, `i_A`, `i_B`, `i_C` e `i_D`, as quais são combinadas por meio de diversas operações lógicas e aritméticas, sendo 4 adições, 4 XORs e 4 rotações, conforme descrito no Quadro 1.

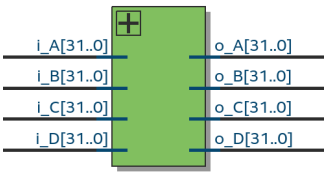

Figura 9: Quarter Round (`QuarterRound`)

Banco de Registradores

O bloco `RegisterFile` (Figura 10) é responsável por armazenar os resultados da função `QuarterRound`, para que estes sejam então escritos de volta no registrador paralelo-paralelo. Este bloco possui quatro entradas, `i_DATA_A`, `i_DATA_B`, `i_DATA_C` e `i_DATA_D` de 32 bits cada, e apenas uma saída de 512 bits.

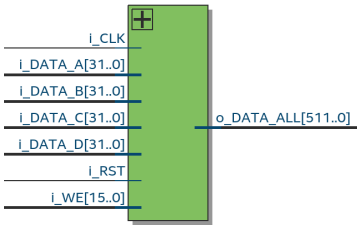

Figura 10: Banco de Registradores (`RegisterFile`)

Contador de Rodadas

O `RoundCounter` (Figura 11) é responsável por armazenar a quantidade de rodadas realizadas até o momento, e sinalizar por meio da saída `o_ALL_ROUNDS_FINISHED` quando todas as 20 rodadas forem concluídas. Contudo, devido a uma escolha de projeto, este elemento conta apenas 10 iterações, incrementado a cada duas rodas inteiras (colunas e diagonais).

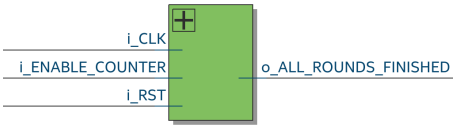

Figura 11: Contador de Rodadas (`RoundCounter`)

Somador Matricial

No bloco `SMAdder` (Figura 12) é realizada a adição matricial de cada um dos 16 elementos da matriz de estado atual (`RegPP`) com a matriz do estado inicial (`Concat`). Na operação de adição, não há bit de transporte (*carry*) entre os elementos da matriz.

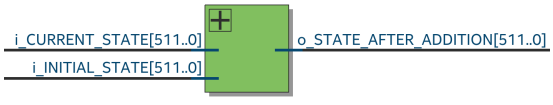

Figura 12: Somador Matricial (`SMAdder`)

Cifrador

Após a realização de todas as rodadas e a soma matricial do estado atual com o estado inicial, é necessário realizar a operação lógica XOR bit-a-bit do texto claro (mensagem a ser cifrada) com o OTP gerado. Essa operação é realizada pelo bloco `XOR` (Figura 13), resultando no texto cifrado na saída `o_CIPHER_TEXT`.

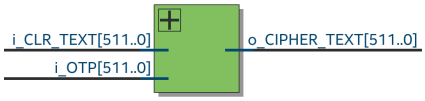

Figura 13: Cifrador (`XOR`)

Controlador

O `Controller` (Figura 14) é responsável por gerenciar o caminho de dados do acelerador, por meio dos seguintes sinais:

- `o_ADDR_A`, `o_ADDR_B`, `o_ADDR_C`, `o_ADDR_D`: sinalizam qual elemento em cada linha do `RegPP` deverá ser lido para uso na rodada atual.
- `o_ENABLE_COUNTER`: habilita o incremento do contador interno de rodadas do `RoundCounter`.
- `o_RDY`: sinaliza que todas as 20 rodadas da cifra foram realizadas e que o resultado está pronto para ser lido.
- `o_SEL_WHICH_REG_PP_INPUT`: seleciona qual entrada do `MUX` será escrita no `RegPP`.
- `o_WE_REG_PP`: habilita a atualização de conteúdo do `RegPP`.
- `o_WE_TO_REGISTER_FILE`: direciona a escrita para uma coluna específica no bloco `RegisterFile`.

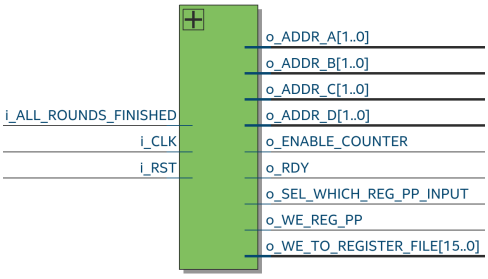

Figura 14: Controlador (`Controller`)

5 RESULTADOS

Esta seção apresenta os resultados de síntese e simulação do acelerador descrito, assim como métricas de desempenho.

5.1 Materiais

Este trabalho utilizou do Intel Quartus Prime 20.1 e do Xilinx Vivado 2020.1 para a síntese do hardware descrito em VHDL. Além disso, foi utilizada a ferramenta ModelSim Intel FPGA 2020.1 para a verificação do comportamento do circuito.

Para a prototipação e testes, foram empregados os kits de desenvolvimento Altera DE1-SoC (Cyclone V) e Xilinx Zedboard (Zynq-7000), ambos contendo um processador ARM Cortex-A9 e uma área de lógica programável em seus FPGAs. A DE1-SoC foi utilizada para a análise da utilização de recursos lógicos por entidade, enquanto a Zedboard foi adotada para a verificação do software executado diretamente no processador (*baremetal*).

A implementação em software utilizou como referência o OpenSSL [12]. Porém, algumas modificações foram necessárias para a execução *baremetal*, como a remoção de macros e a lógica de conversão entre *little-endian* e *big-endian*, além da confecção de um novo ponto de entrada do programa.

5.2 Síntese

A Tabela 2 apresenta a quantidade de recursos lógicos utilizados pelo acelerador, em termos de Look-Up Tables (LUTs) e Flip-Flops (FFs), e a frequência máxima de operação (Fmax) em cada um dos dispositivos utilizados.

Tabela 2: Recursos lógicos utilizados pelo acelerador

Dispositivo	LUTs	FFs	Fmax (MHz)
Altera Cyclone V	1429	1039	62,49
Xilinx Zynq-7000	1573	1044	82,57

A síntese para o Xilinx Zynq-7000 utiliza mais recursos que no Altera Cyclone V, com um aumento significativo na frequência máxima de operação. Na Tabela 3 é apresentada a utilização dos recursos de cada componente, utilizando como referência a síntese para o Altera Cyclone V. O componente `Concat` não é listado por ser composto apenas pelo agrupamento de fios.

Tabela 3: Utilização de recursos lógicos por componente

Componente	LUTs	FFs
`MUX`	56	0
`RegPP`	0	512
`Selector`	128	0
`QuarterRound`	192	0
`RegisterFile`	0	512
`RoundCounter`	5	4
`SMAdder`	507	0
`XOR`	512	0
`Controller`	29	11

5.3 Simulação

Para a verificação do projeto foram desenvolvidos *testbenches* visando confirmar o correto funcionamento de cada bloco individualmente, com posterior verificação do acelerador completo. Pela análise das formas de onda foi possível observar a correta execução de todas as etapas da cifra, desde a inicialização da matriz de estado inicial, função Quarter Round, soma matricial e a cifragem do texto claro.

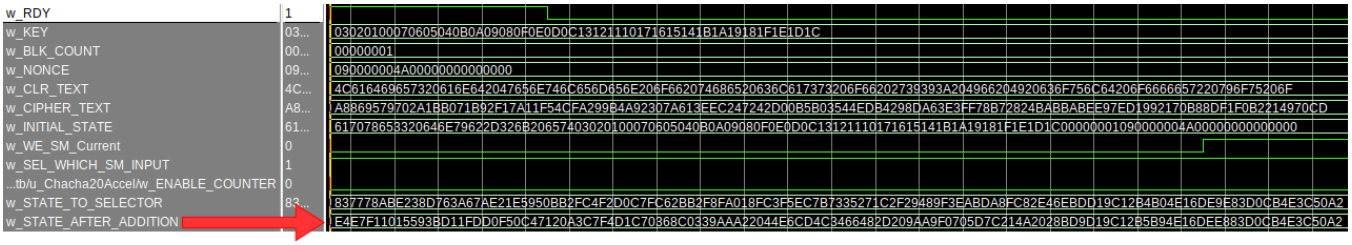

Figura 15: Diagrama de formas de onda após a execução das 20 rodadas

Na Figura 15 é apresentado o diagrama de formas de onda após a execução das 20 rodadas. O sinal `o_RDY` está no nível lógico alto e o resultado do estado da cifra (`w_STATE_AFTER_ADDITION`) é idêntico ao exemplo apresentado na seção 2.3.2 da RFC 8439 (Quadro 3), comprovando assim o correto funcionamento do acelerador desenvolvido.

Quadro 3: Gabarito do exemplo apresentado na RFC 8439

E4E7F110	15593BD1	1FDD0F50	C47120A3
C7F4D1C7	0368C033	9AAA2204	4E6CD4C3
466482D2	09AA9F07	05D7C214	A2028BD9
D19C12B5	B94E16DE	E883D0CB	4E3C50A2

Fonte: RFC 8439 [7]

5.4 Avaliação de Desempenho

Considerando a quantidade de ciclos necessárias para cifrar 512 bits de dados no acelerador desenvolvido (1015 ciclos) e no ARM Cortex-A9 presente no Xilinx Zynq-7000 (3545 ciclos), e a frequência máxima de operação de cada dispositivo, é possível derivar a vazão de cada solução (Tabela 4).

Tabela 4: Vazão obtida nas implementações

Dispositivo	Fmax (MHz)	Ciclos	Vazão (Mb/s)
Altera Cyclone V	62,49	1015	31,52
Xilinx Zynq-7000	82,57	1015	41,65
ARM Cortex-A9	667,00	3545	96,33

Apesar da implementação em software possuir uma vazão maior, é preciso considerar a frequência de operação de cada plataforma. De forma a medir a aceleração obtida pela implementação em hardware, a frequência do processador foi normalizada para a frequência de operação do acelerador, obtendo assim o ganho efetivo de desempenho da implementação em hardware.

O acelerador prototipado no Xilinx Zynq-7000 é 3,49 vezes mais rápido se comparado a implementação em software operando na mesma frequência (82,57 MHz), conforme apresentado na Tabela 5.

Tabela 5: Aceleração normalizada

Implementação	Ciclos	Vazão (Mb/s)	Aceleração
Software	3545	11,92	-
Hardware	1015	41,65	3,49x

6 CONCLUSÃO

Neste trabalho foi apresentado o desenvolvimento de um acelerador de criptografia de fluxo baseado na cifra ChaCha20, amplamente utilizada para a proteção do fluxo de dados.

O trabalho demonstrou os resultados obtidos em duas plataformas de avaliação distintas, relatando a utilização de recursos lógicos e o desempenho de cada implementação. Os resultados foram comparados com uma implementação em software, obtendo uma aceleração normalizada de 3,49x no acelerador desenvolvido.

Como trabalhos futuros, pretende-se implementar o código autenticador Poly1305 [13], bem como otimizar o desempenho do acelerador por meio de *pipelining* e da replicação do bloco `QuarterRound`. Além disso, espera-se integrar o acelerador desenvolvido em um SoC (System-on-Chip).

AGRADECIMENTOS

Este trabalho foi financiado em parte pela Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) e pela Fundação de Amparo à Pesquisa e Inovação de Santa Catarina (FAPESC), termos 2021TR001907 e 2021TR001236.

REFERÊNCIAS

[1] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners.* Springer Science & Business Media, 2009.

[2] Nirav Shah Fortinet. The challenges of inspecting encrypted network traffic, Aug 2020. URL https://www.fortinet.com/blog/industry-trends/keeping-up-with-performance-demands-of-encrypted-web-traffic.

[3] Frank Vahid and Tony Givargis. Platform tuning for embedded systems design. *Computer*, 34(2):112–114, 2001.

[4] A.J. Menezes, J. Katz, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography.* Discrete Mathematics and Its Applications. CRC Press, 1996. ISBN 9781439821916. URL https://books.google.com.br/books?id=MhvcBQAAQBAJ.

[5] Luiz Gonzaga de Alvarenga. *Criptografia Clássica e Moderna.* Clube de Autores, 2011.

[6] Daniel J Bernstein et al. Chacha, a variant of salsa20. In *Workshop record of SASC*, volume 8, pages 3–5, 2008.

[7] Adam Langley Yoav Nir. Chacha20 and poly1305 for ietf protocols. RFC 4180, IRTF, 03 2018. URL https://datatracker.ietf.org/doc/html/rfc8439.

[8] Robson Sopran, Douglas R Melo, Cesar A Zeferino, and Eduardo A Bezerra. Análise comparativa do custo e do desempenho de um algoritmo de criptografia para sistemas embarcados explorando o particionamento hardware/software. *Anais do Computer on the Beach*, pages 259–268, 2017.

[9] Sachin Kumar, Jawad Haj-Yahya, and Anupam Chattopadhyay. Efficient hardware accelerator for norx authenticated encryption. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2018.

[10] Islam Mohamed Shaher, Moustafa Mahmoud, Hassan Ibrahim, Moustafa Ali, and Hassan Mostafa. Implementation of a hardware accelerator for a real-time encryption system. In *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 627–630, 2020.

[11] Igor Semenov. *An Implementation of ChaCha20 Stream Cypher in All-programmable SoCs.* PhD thesis, The University of Alabama in Huntsville, 2020.

[12] The OpenSSL Project Authors. Openssl, 2022. URL https://github.com/openssl/openssl/blob/1c0eede9827b0962f1d752fa4ab5d436fa039da4/crypto/chacha/chacha_enc.c.

[13] Daniel J Bernstein. The poly1305-aes message-authentication code. In *International workshop on fast software encryption*, pages 32–49. Springer, 2005.